Operativsystem ID2200/06

omtentamen

2017-06-07 8:00-12:00

Instruktioner

- Du får, förutom skrivmateriel, endast ha med dig en egenhändigt handskriven A4 med anteckningar. Mobiler etc skall lämnas till tentamensvakterna.
- Svaren skall lämnas på dessa sidor, använd det utrymme som finns under varje uppgift för att skriva ner ditt svar.
- Svar skall skrivas på svenska eller engelska.
- Du skall lämna in hela denna tentamen.
- Inga ytterligare sidor skall lämnas in.

Versioner

Denna tentamen gäller för flera olika omgångar av kurserna ID2200/06. Beroende på vilken kursomgång du följer så skall olika delar av tentamensfrågorna besvaras.

För omtentander i ID2200 gäller följande:

- Registrerade för tentamen på 6hp, VT16 och HT16: besvara frågorna 1-9, inte fråga 10.
- Registrerade för tentamen på 3.8 hp, dvs före VT16: besvara frågorna 1-8, inte 9-10.
- För de som är registrearde för tentamen på 3.8hp men som ännu inte har lab-momentet avklarat kan man besvara även fråga 9 och då få det momentet tillgodoräknat. Fråga 9 hanteras separat så få poäng på fråga 9 kompensears inte av flera poäng i övriga delar.

För omtentander i ID2206 gäller följande:

- Registrerade för tentamen på 6hp, HT16: besvara frågorna 1-9, inte 10.
- Registrerade för tentamen på 4.5hp, före HT16: besvara frårna 1-8 och fråga 10
- För de som är registrearde för tentamen på 4.5hp men som ännu inte har lab-momentet avklarat kan man besvara även fråga 9 och då få det

momentet delvis tillgodoräknat. Fråga 9 hanteras separat så få poäng på fråga 9 kompenseras inte av flera poäng i övriga delar.

Betyg för 6hp

Tentamen har ett antal uppgifter där några är lite svårare än andra. De svårare uppgifterna är markerade med en stjärna, *poäng**, och ger poäng för de högre betygen. Vi delar alltså upp tentamen i grundpoäng och högre poäng. Se först och främst till att klara grundpoängen innan du ger dig i kast med de högre poängen.

Notera att det av de 40 grundpoängen räknas bara som högst 36 och, att högre poäng inte kompenserar för avsaknad av grundpoäng. Gränserna för betyg är som följer:

- Fx: 21 grundpoäng
- E: 23 grundpoäng
- D: 28 grundpoäng
- C: 32 grundpoäng
- B: 36 grundpoäng och 12 högre poäng
- A: 36 grundpoäng och 18 högre poäng

Gränserna kan komma att justeras nedåt men inte uppåt.

Gränsen för E är för tentamen på 4.5hp 18 poäng och för 3.8hp tentamen 16 poäng. Gränsen för tillgodoräkning av lab-moment är 12 poäng på fråga 9.

Namn:_____ Persnr:_____

1 Operativsystem

1.1 vad händer här? [2 poäng]

Om vi ger kommandona nedan, efter varandra, i ett *shell*; vad kommer resultat att vara?

```
> mkdir foo
> echo "gurka" > foo/gronsak.txt
> ln -s gronsak.txt foo/check.txt
> mkdir bar
> echo "morot" > bar/gronsak.txt
> mv foo/check.txt bar
> cd bar
> cat check.txt
```

Svar: Resultatet blir att `morot` skrivs ut eftersom `check.txt` är en mjuk länk.

1.2 kommandon i ett shell [2 poäng]

Ge en kort beskrivning av vad kommandona nedan gör.

- `chmod`
- `diff`
- `sed`
- `tail`

Svar: Slå upp deras betydelse med hjälp av `man`.

Namn:_____ Persnr:_____

2 Processer

2.1 vad är var? [2 poäng]

I koden nedan har vi variablerna: `x`, `i`, `h` och `c`. I vilka segment återfinns de datastrukturer som de är bundna till: globalt, stack eller heap? Vad kommer skrivas ut på stdout?

```
#include <stdio.h>
#include <stdlib.h>

int x = 3;

int foo(int i) {
  int h[] = {1,2,3,4};
  return h[i];
}

int main() {

  int c = foo(x);
  printf("h[%d] = %d \n", x, c);
  return 0;
}
```

Svar: `x` är global, `h[]`, `i` och `c` ligger på stacken. Utskriften blir: `h[3] = 4`.

2.2 IDT:n [2 poäng*]

I en x86-arkitektur så finns en IDT som vi använder när vi implementerar bland annat systemanrop. Vad måste operativsystemet lägga in i IDT:n för att möjliggöra ett systemanrop?

Svar: Operativsystemet lägger in en pekare på en bestämd position i tabellen (0x80) till en procedur som tar hand om alla systemanrop. När en användarprocess exekverar `INT 0x80` kommer den proceduren att få kontroll och kommer då att exekvera i s.k. *kernel mode*.

Namn:_____ Persnr:_____

3 Schemaläggning

3.1 multi-level feedback queue [2 poäng]

När vi implementerar en schemaläggare med en s.k. *multi-level feedback queue* så kan vi använda en strategi för att ge högre prioritet till interaktiva processer. Hur ser den strategin ut?

Svar: Processer som inte utför en I/O operation under sin tilldelade tid kommer att flyttas ned till en lägre prioritetsnivå. För att processer inte skall fastna i lägre prioritetsnivåer får man med jämna mellanrum lägga upp processer till den översta prioritetsnivån.

3.2 lotteri [2 poäng]

I s.k. lotteribaserade schemaläggare så är inte fokus att minimera vare sig reaktionstiden eller omloppstiden. Vad är det vi försöker uppnå och hur kan vi uppnå det genom att implementera ett lotteri?

Svar: Vi vill att varje process skall få snin beskärda del av resurserna. Vi uppnår detta genom att tilldela varje process en mängd lotter som motsvara dess beskärda del; några får fler andra mindre.

3.3 multiprocessor [2 poäng*]

När man implementerar en schemaläggare för en multiprocessor så har man ytterligare en viktig sak att ta hänsyn till. Vad måste man ta hänsyn till och vilken strategi bör man använda för att hantera problemet?

Svar: Det kostar att flytta en process från en processor (eller kärna) till en annan. Man bör i möjligast mån låta jobb köra på samma processor. Endast om det blir en stor obalans så skall man börja flytta processer.

4 Virtuellt minne

4.1 paging [2 poäng]

Antag att vi har en virtuell adress som består av ett 20-bitars sidnummer och en 12-bitars offset. Vi har en fysisk adressrymd på hela 36 bitar vilket gör vår omvandling från virtuella adresser till fysiska adresser intressant. Vad

kommer vår omvandlingstabell att innehålla och hur mycket minne kan en enskild process adressera?

Svar: Sidorna är 4Kibyte stora eftersom vi har 12 bitars offset. Det betyder att vi kan ha 2^{24} ramar i minnet. Vår omvandlingstabell kommer nu att mappa sidnummer på 20 bitar till ramnummer på 24 bitar. Varje enskild process har fortfarande en begränsning på 32 bitar dvs 4 GiByte.

4.2 TLB [2 poäng]

En TLB är viktig för att adressomvandling skall gå tillräckligt snabbt.Vad är en TLB och hur snabbar den upp omvandlingen?

Svar: En TLB är en cache för omvandlingselement. Istället för att gå ut till minnet en eller flera gånger för att hämta rätt element i en tabell så kan vi lagra dessa i en cache. Vid adressomvandling kan TLB:n adresseras med det sidnummer vi söker och om vi får en träff får vi direkt tillgång till det sökta ramnummret.

4.3 x86_64 [2 poäng*]

I en x86_64 arkitektur har vi i en virtuell adress på 48 bitar. Den är uppdelad i ett sidnummer kodat med fyra segment med 9 bitar var och ett offset på 12 bitar. Varför är sidnummret uppdelat i fyra delar och på 9 bitar var? Varför inte tre eller två segment, eller varför inte bara ha ett enda värde på 36 bitar?

Svar: Uppdelningen har med vår önskan att implementera omvandlingstabellen i en trädstruktur. Om varje segment är på 9 bitar så kan de indexera 512 element vilket ryms i en sida på 4KiByte om varje element är på 8 bytes. Vi kommer att behöva 8 byte eftersom bara ett ramnummer på 40 bitar tar upp 5 byte. Om vi väljer färre segment så blir varje segment större (12 eller 18) bitar och dessa tabeller ryms inte i en sida vilket komplicerar saker. Om vi bara har ett segment så har vi inte ens en trädstruktur och tabellen blir helt platt vilket tar upp mycket utrymme.

5 Minneshantering

5.1 intern och extern fragmentering [2 poäng]

Ge en förklaring på vad som menas med intern och extern fragmentering.

Svar: Vid intern fragmentering har vi delat ut resurser som inte till fullo används; resurserna är oanvända men de kan inte ges till någon annan. Vid extern fragmentering har vi en mängd resurser som är för små för att möta den efterfråga som finns. Resurserna är utspridda och kan inte på något enkelt sätt slås ihop till större resurser.

5.2 best-fit vs worst-fit [2 poäng]

Vad skulle fördelen vara att istället för att välja den resurs som bäst matchar en förfrågan (best-fit), välja den som är så stor som möjligt (worst-fit)?

Svar: Vid best-fit så blir den resterande delen mycket liten. Vi riskerar att dela upp våra resurser i mindre och mindre delar vilket ökar den externa fragmenteringen. Vid worst-fit så kommer den återstående resursen att vara betydligt större vilket förhoppningsvis har minskar fragmenteringen.

Namn:_____ Persnr:_____

5.3 segregerade listor [2 poäng*]

En strategi för att implementera hanteringen av s.k. *free-listor* i en minneshanterare är att låta alla minnesblock vara av storleken av en två-potens (med något minsta värde t.ex. 32 byte). Om ett block av önskad storlek inte finns så tar man ett block av närmast större storlek och delar det på två. När man lämnar tillbaks ett block så vill man kanske kontrollera om man kan sammanfoga detta med ett annat fritt block för att på så sätt undvika att det bildas fler och fler små block. Hur skall vi enkelt kunna hitta det block som vi om möjligt kan sammanfoga ett frigjort block med? Har strategin några begränsningar?

Svar: Vi kan använda en buddy-strategi vilket gör att vi snabbt kan avgöra om en hopslagning är möjlig. Detta har dock begränsningen att t.ex. två angränsande fria block på 64 bytes kanske inte kan sammanfogas eftersom de inte är "buddies". Vi har även en intern fragmentering eftersom vi bara delar ut block som är 2-potenser stora.

Namn:_____ Persnr:_____

6 Flertrådad programmering

6.1 count [2 poäng]

C-koden nedan, med motsvarande assembler, pekar på något vi måste ta hänsyn till när vi använder delat minne för trådar. Vilket problem är det som koden pekar ut?

```
int loop = 10;
int count = 0;

void *hello(void *) {
      :
  for(int i = 0; i < loop; i++) {
      count++;
  }
      :
}
```

```
.L3:
        movl    count(%rip), %eax
        addl    $1, %eax
        movl    %eax, count(%rip)
        addl    $1, -4(%rbp)
        movl    loop(%rip), %eax
        cmpl    %eax, -4(%rbp)
        jl      .L3
```

Svar: Operationen ++ är inte atomär utan består av en läsning, en addering och en skrivning. Om en tråd blir avbruten, eller om två trådar körs parallelt, så kan resultatet bli annorlunda än om de kör efter varandra.

6.2 Anna Book [2 poäng]

Det finns en enkel strategi som man ibland kan följa för att garanterat undvika att ett deadlock uppstår. Vi kan alltså helt undvika situationen, inte bryta oss ur ett deadlock som redan har uppstått. Beskriv denna strategi och förklara varför den undviker dead-lock.

Svar: Vi kan ordna resurser i en total ordning och alltid begära resurser i denna ordning. Om vi tvingas vänta på en resurs så hålls den av en process som antingen exekverar eller väntar på en resurs med högre ordning. Vi undviker därmed ett cirkulärt beroende.

6.3 TSFO [2 poäng*]

Du och din kollega Alan Peterson har insett att ni behöver implementera ett lås för att skydda delade datastrukturer. Alan föreslår att varje tråd har en flagga för att marker att de vill gå in i en kritisk sektion. Om en tråd först sätter sin egen flagga och sen kontrollerar att ingen annan tråd har satt sin

flagga så har man löst problemet. Om någon annan tråd också har satt sin flagga så nollställer man sin egen och prövar lite senare. En risk finns för *starvation* men det är ett problem som man kan ta.

Vad säger du till din kollega?

Svar: Total Store Order - en modern processor ger oss inte sekvensiell konsistens vilket skulle behövas för att algoritmen skall fungera. Eftersom bara TSO erbjuds kan vi få en situation där flera trådar är inne i den kritiska sektorn samtidigt.

Namn:_____ Persnr:_____

7 Filsystem och lagring

7.1 länkar [2 poäng]

Om man skall implementera en traversering av en generell graf så måste man se upp med att inte hamna i en cirkulär struktur. Om du ombads söka igenom en mapp och alla dess undermappar i ett Unix-system, där du vet att det förekommer en hel del både hårda och mjuka länkar, vilken strategi skulle du använda för att undvika hamna i en loop?

Svar: De hårda länkarna bildar, om vi bortser från mjuka länkar och de speciella länkarna `.` och `..`, ett träd. En strategi kan vara att endast gå på de hårda länkarna och ignorera de mjuka. Eftersom dessa bildar ett träd kan vi söka igenom mapparna utan att riskera hamna i en loop.

7.2 inoder och datablock [2 poäng]

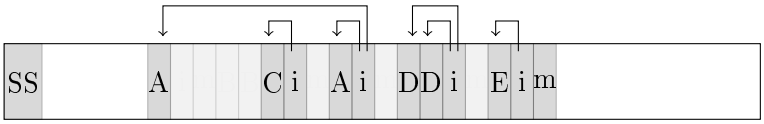
Ett filsystem representerar filer och mappar med hjälp av inoder och datablock. Antag att vi inte har någonting i någon fil-cache utan måste läsa allt från disk. Vilka inoder och datablock måste vi då läsa för att börja kunna läsa från filen `/home/alan/gurka.txt`? Lista inoderna och blocken i den ordning vi kommer läsa dem.

Svar: Följande inoder och block måste läsas: inod för rot (`/`), datablock för rot, inod för `home`, datablock för `home`, inod för `alan`, datablock för `alan`, inod för `gurka.txt` och till sist datablock för filen.

Namn:_____ Persnr:_____

7.3 loggbaserat filsystem [2 poäng*]

Nedan ser du en schematisk bild av ett loggbaserat filsystem. Om systemet nu börjar få ont om plats så kommer det att försöka skapa plats. Hur kan mer plats skapas och hur kommer systemet att se ut efter det att mer utrymme tillgängliggjorts?



Svar: Mer plats skapas genom att filer som ockuperar block i systemet "svans", kopieras till dess huvud. I exemplet ovan kan man visa att filen `A` har kopierats och därmed frigör konsekutivt minne fram till blocket som används för filen `C`.

Namn:_____ Persnr:_____

8 Virtualisering

8.1 kernel mode [2 poäng]

Antag att vi har en processor som enbart erbjuder två exekveringsnivåer: *user mode* och *kernel mode*. Vid virtualisering av ett helt operativsystem måste vi låta det virtualiserade operativsystemet köra i kernel mode - sant eller falsk? Motivera ditt svar.

Svar: Falskt, operativsystemet skall köra i *user mode*. Alla avbrott kommer att kontrolleras av den *hypervisor* som övervakar de virtualiserade operativsystemen.

8.2 containers [2 poäng*]

Så kallade "Linux containers" är också ett sätt att köra flera operativsystem på samma maskin. Vilken begränsning har denna metoden jämfört med full virtualisering?

Svar: Vi är begränsade till att köra samma operativsystem som operativsystemet som kör på vår värdmaskin. Alla virtuella system använder samma kärna.

Namn:_____ Persnr:_____

9 Implementering

9.1 danger ahead [2 poäng]

Det är inte helt definierat vad som kommer hända när vi kör koden nedan. Vad är det vi gör fel och vad kan en möjlig effekt bli?

```
int main() {

  char *heap = malloc(20);
  *heap = 0x61;
  printf("heap pointing to: 0x%x\n", *heap);
  free(heap);

  char *foo = malloc(20);
  *foo = 0x62;
  printf("foo pointing to: 0x%x\n", *foo);

  *heap = 0x63;
  printf("foo pointing to: 0x%x\n", *foo);

  return 0;
}
```

Svar: Vi frigör den area som pekas ut av `tt heap` men sen använder vi den i alla fall sex rader senare. Det är odefinierat vad som händer men i detta fall så kan det bli så att den frigjorda arean delas ut när vi anropar `malloc(20)` igen. När vi då skriver på `*heap` så förandrar vi vad `foo` pekare på.

Namn:_____ Persnr:_____

9.2 dup2() [2 poäng]

Antag att vi har ett program `boba` som skriver "Don't get in my way" på `stdout`. Vad kommer resultatet bli om vi kör programmet nedan och varför blir det så? (proceduren `dprintf()` tar en fildescriptor som argument)

```
int main() {

  int fd = open("quotes.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);

  int pid = fork();

  if(pid == 0) {
    dup2(fd, 1);
    close(fd);
    execl("boba", "boba", NULL);
  } else {
    dprintf(fd, "Arghhh!");
    close(fd);
  }
  return 0;
}
```

Svar: I `dup2(fd,1)` sätter vi om `stdout` till den öppnade filen. Boba kommer då att skriva sin rad till filen `quotes.txt`. Samtidigt kommer moderprocessen att skriva "Arghhh!" på samma fil. De två processerna kommer dock att dela på samma representation av den öppnade filen, dela position och samsas om att skriva. Vi kommer därför se en blandning av de båda texterna i filen `quotes.txt` i.e. den ena texten kommer inte skriva över den andra.

Namn:_____ Persnr:_____

9.3 ctrl-c [2 poäng]

Ett enkelt sätt att döda ett program är att trycka ctrl-c. Om vi skriver ett program så kanske vi inte vill dö eller vi kanske vill utföra några sista operationer innan vi terminerar. Vilka mekanismer skall vi använda i vårt program för att hantera detta?

Svar: Vi skall skriva en liten signalhanterare, en procedur som vi registrerar som hanterare för en speciell signal, i detta fall `SIGINT`. Ett `ctrl-c` kommer att resultera i att `SIGINT` skickas till processen och det kommer då i sin tur resultera i att proceduren körs.

9.4 slump inte helt fel [2 poäng]

Om vi implementerar en procedur som skall kasta ut en sida ur minnet när det är fullt så kan vi plocka en sida helt slumpmässigt. Om vi har ett minne som består av r ramar och vi har totalt n sidor som skall samsas i minnet så kan man tänka sig att den slumpmässiga strategin skulle ge oss en sannolikhet för träff på n/r. I verkligheten får vi dock oftast ett resultat som är bättre, vad beror det på?

Svar: Vi har troligtvis inte en slumpmässig användning av sidorna. Vi har troligtvis både en temporär och rumslig lokalitet vilket gör att vi har en större chansa att hitta de sidor som vi söker i minnet.

Namn:_____ Persnr:_____

9.5 lista innehållet i en map [2 poäng]

Om vi vill lista innehållet i en map så kan vi använda oss av biblioteksrutinen `opendir()`. Vilken information kan vi direkt få ut av strukturen som pekas ut av `entry` i koden nedan? Vilka egenskaper av en fil kan vi inte hitta direkt och var kan vi hitta dessa?

```
int main(int argc, char *argv[]) {

  char *path = argv[1];

  DIR *dirp = opendir(path);

  struct dirent *entry;

  while((entry = readdir(dirp)) != NULL) {

    //  what information do we have?

  }
}
```

Svar: Det vi kan hitta direkt är filernas namn, dess typ och dess inodnummer. För alla andra egenskaper (storlek, skapad, ändrad, ägare etc) måste vi hämta in filens inod.

9.6 läshastighet [2 poäng]

Antag att vi har en vanlig hårddisk som är kopplad via en SATA-förbindelse på 6 Gb/s vilket ger oss en lästid av ett slumpvist 4 KByte block på 12 ms. Hur förändras detta om byter ut vår disk till en som vi sätter upp med en SAS-förbindelse på 12 Gb/s? Motivera.

Svar: Tiden för att läsa ett 4KByte block domineras av tiden för att röra armen och rotera disken till rätt position Den nya hårddisken kan ha bättre värden för dessa egenskaper men att förbindelsen ökar i kapacitet ger en marginell skillnad för enskilda block.

9.7 ett huvud och en fotnot [2 poäng*]

Vid implementation av minnesallokering så är det vanligt att man ha ett gömt *huvud* placerat alldeles innan den minnesarea som man delar ut. I detta huvud kan man bland annat skriva hur stor arean är så att det blir

enklare att ta hand om arean när vi gör *free*. Man kan även använda sig av en gömd fotnot som ligger efter arean där man kan skriva att arean är använd eller inte och kanske en pekare till huvudet. Vad är det för poäng med att lägga den informationen efter arean, räcker det inte med huvudet?

Svar: När vi gör free på en area kan vi titta i fotnoten på den area som ligger alldeles innan den area som vi vill göra free på. Om den arean är fri så kan vi kanske slå ihop blocken till ett större block. Vi kan som vanligt titta i arean framför oss, där är det ingen skillnad.

9.8 lite bättre [2 poäng*]

Så kallade *pipes* är ett mycket enkelt sätt att skicka data från en process till en annan. Det har dock sina begränsningar och ett bättre sätt är att använda s.k. *sockets*. Om vi istället för en pipe öppnar en *stream socket* mellan två processer så har vi flera fördelar. Beskriv två saker som en *stream socket* ger oss som vi inte får om vi använder en *pipe*.

Svar: En *stream socket* ger oss en dubbelriktad kanal där vi kan välja flera olika addresseringsmetoder. Om vi använder *pipes* är vi begränsade till filsystemet för adressering, om vi använder sockets kan vi adressera en process på en annan dator.

Namn:_____ Persnr:_____

9.9 namnrymd [2 poäng*]

Nedan är kod där vi öppnar en socket och använder namnrymden `AF_INET`. Vi kan då adressera en server med hjälp av portnummer och IP-adress. Det finns en annan namnrymd som vi kan använda när vi arbetar med socket. Nämn en och beskriv vilka för och nackdelar den kan ha.

```
struct sockaddr_in server;
server.sin_family = AF_INET;
server.sin_port = htons(SERVER_PORT);
server.sin_addr.s_addr = inet_addr(SERVER_IP);
```

Svar: Vi kan använda filnamn som namnrymd. Detta har nackdelen att vi inte kan nås från andra noder i ett nätverk utan är begränsade till den nod vi kör på. Det kan naturligtvis också vara en fördel då vi slipper exponera oss för omvärden. Implementeringen kan också vara mer effektiv eftersom vi inte behöver använda oss av t.ex. TCP.

9.10 AC/DC [2 poäng*]

Om vi har allokerat en två-dimensionell array `table` och som vi sedan vill summera alla värden ur så kan vi göra det enligt koden nedan. Vad i koden nedan får ett oönskat beteende och hur skulle vi kunna förbättra körtiden? Motivera.

```
#define ROWS 4000
#define COLS 1000

int table[ROWS][COLS];

int main() {

  :
  long sum = 0;

  for(int c = 0; c < COLS; c++) {
     for(int r = 0; r < ROWS; r++) {
       sum += table[r][c];
     }
  }
  :

}
```

Namn:_____ Persnr:_____

Svar: Problemet är att vi kommer att gå igenom matrisen kolumn för kolumn men den ligger rad för rad i minnet. Varje rad är 4 KByte (1000 int) så vi kommer att hoppa från sida till sida för när vi går från en rad till en annan. Om vi istället traverserar matrisen rad för rad så kommer vi att stanna inom samma sida och gå igenom alla element i raden innan vi går till nästa sida. Detta kommer ha inverkan på TLB:n och ge oss en bättre körtid. Prova gärna det är rätt så stor skillnad.

Namn:_____ Persnr:_____

10 Bara för omtentamen i ID2206 reggade före HT16 (4.5hp)

10.1 Coffmans villkor [2 poäng]

Coffman beskrev fyra villkor som måste gälla om vi skall kunna får ett deadlock. Gen en kortfattad beskrivning av dessa villkor.

Svar:

- ömsesidig uteslutning: det finns resurser som endast kan hållas av ett begränsat antal resurser
- behåll och vänta: processer kan hålla en resurs medan de väntar på en annan
- icke avbrytbar: har vi väl börjat vänta på en resurs så kan vi inte avbryta
- cirkulärt beroende: processer väntar på resurser som hålls av andra processer i en cirkulär struktur

10.2 Earliest Deadline First [2 poäng]

Vid schemaläggning i ett realtidssystem så kan man använda sig av den enkla strategin att ha en avbrytande schemaläggare och alltid schemalägga den process som har minst tid kvar till sin dead-line. Hur bra fungerar denna schemaläggare?

Svar: Den är så bra att den kan garantera att alla jobb hinner bli utförda om det överhuvudtaget går.

Namn:_____ Persnr:_____

10.3 Lamport hash [2 poäng*]

I Unix-system lagras oftast användares hashade lösenord (med ett salt) vilket oftast är fullt tillräckligt. Man kan bygga ett säkrare system med ett så kallat *Lamport hash*. Här lagrar servern ett hashvärde av lösenordet $h^{i+1}(paswd)$ och ett värde i. När användaren loggar in kan servern begära hashvärdet för $h^i(paswd)$ som bevis på att användaren känner till lösenordet. Servern kan själv genererar $h(reply)$ eftersom funktionen är känd. Om resultatet överensstämmer med $h^{i+1}(paswd)$ så uppdaterar servern sitt tillstånd till $h^i(paswd)$ och $i-1$ och kan fortsätta tills $i = 1$. Vad är fördelen med denna metod?

Svar: En avlyssnare som hör $h^i(paswd)$ kan inte använda det i en attack eftersom servern nästa gång kommer att begära $h^{i-1}(paswd)$. Man inte skapa $h^{i-1}(paswd)$ även om man har kunskap om $h^i(paswd)$.

10.4 RPC [2 poäng*]

Vid implementering av RPC kan man välja mellan att erbjuda “at least once” eller “at most once” vad betyer dessa begrepp?

Svar: Vid “at least once” kommer systemet inte att skicka om en förfrågan som kanske har gått förlorad vilket gör systemet mer tillförlitligt men vi riskera o andra sidan att en förfrågan hanteras mer än en gång. Vid “at most once” kommer systemet inte att skicka om en förfrågan för att därmed undvika problemet.